VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Kategorizace a vyhledávání fotografií

Categorization and Search of Photos

2014

Michal Šimon

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Michal Šimon**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Kategorizace a vyhledávání fotografií
Categorization and Search of Photos

Zásady pro vypracování:

Vyhledávání fotografií podle podobnosti je velmi užitečný nástroj, který vídáme v mnoha moderních internetových vyhledávačích. Cílem této práce je prostudovat metody pro kategorizaci fotografií podle podobností a navrhnout vlastní implementaci jednoduchého vyhledávače.

1. Nastudovat metody kategorizace a měření podobnosti fotografií.
2. Vybrat nebo navrhnout metodu pro kategorizaci.
3. Naimplementovat webovou aplikaci plnící funkci kategorizace a vyhledávání podobných fotografií.
4. Ověřit a kriticky zhodnotit funkci systému.

Seznam doporučené odborné literatury:

Gonzalez, Rafael C. and Woods, Richard E., "Digital Image Processing (3rd Edition)", 2006, ISBN 013168728X, Prentice-Hall, Inc., Upper Saddle River, NJ, USA

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Karel Mozdřeň**

Datum zadání: 01.09.2013
Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.*

V Ostravě 6. května 2014 Michal Šimon

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 6. května 2014 Michal Šimon

Rád bych na tomto místě poděkoval svému vedoucímu Ing. Karlu Mozdřeňovi za jeho ochotu a trpělivost. Dále bych rád poděkoval své rodině a přátelům za morální podporu a v neposlední řadě také své přítelkyni za trpělivost a psychickou podporu.

Abstrakt

Tato práce popisuje teoretické i praktické řešení úlohy porovnávání obrazů. První a druhá kapitola se věnuje teoretickému základu analýzy obrazu a existujícím metodám řešení této úlohy. Ve třetí kapitole je detailněji popsán postup při tvorbě reálné webové aplikace pomocí HTML5 s ohledem na mobilní zařízení. Věnuji se také popisu nastavení a použití konkrétních nástrojů a frameworků. Z experimentů si lze udělat obrázek nejen o náročnosti a specifičnosti této úlohy, ale nahlédneme také do několika způsobů optimalizace a urychlení některých částí aplikace.

Klíčová slova: Analýza obrazu, JavaScript, Python, Google Closure, Grunt, Bower, NPM, CoffieScript

Abstract

This thesis describes the theoretical and practical solution of the problem comparing images. The first and second chapter deals with the theoretical basis of image analysis and existing methods for solving this task. The third chapter describes in detail the procedure for creating real web applications using HTML5 with regard to mobile devices. I also deal with the description of the setting and the use of specific tools and frameworks. Through experiments you can make a picture not only of the complexity and specificity of this task, but also a glimpse into several ways of optimizing and speeding up some parts of the application.

Keywords: Image analysis, JavaScript, Python, Google Closure, Grunt, Bower, NPM, CoffieScript

Seznam použitých zkratek a symbolů

MVC	–	Model View Controller je architektura návrhu aplikace.
JS	–	JavaScript je programovací jazyk používaný především pro tvorbu webových aplikací.
JSON	–	JavaScript Object Notation je způsob reprezentace strukturovaných dat.
NPM	–	Node Packaged Modules je balíčkovací systém pro platformu Node.JS
PaaS	–	Platform as a Service je služba, u které poskytovatel zajišťuje kompletní prostředky pro tvorbu a běh webové aplikace.

Obsah

Seznam tabulek

Seznam obrázků

1 Úvod

Fotografie, od svého vzniku, prošla velkou evolucí a v dnešní době je její využití díky digitalizaci velmi široké. Aplikací algoritmů analýzy obrazu na digitální fotografii se člověku otevírá obrovský prostor k využití těchto technologií nejen v průmyslu, ale i v každodenním životě. Z tohoto důvodu jsem si vybral jako téma své práce část této problematiky a pokusil se zpracovat své řešení problému kategorizace a vyhledávání fotografií. V kapitole 2 se budu zabývat obecnými postupy řešení a v kapitole 3 detailně rozeberu implementaci a zvolené technologie. Kapitola 4 pak uzavírá celou práci popisem experimentů a optimalizací.

1.1 Analýza obrazu

Počítačová grafika se zabývá vizualizací a modelováním reálných objektů. Využívá k tomu rasterizaci a další aproximační techniky tak, aby uživateli co nejlépe graficky popsala dané objekty. Naproti tomu se analýza obrazu zabývá procesem zcela opačným. Z obrazu se snaží získat zpět co nejvíce informací o tom, co se na obraze nachází.

Analýza obrazu často vychází z principů fungování lidského oka a také z fyzikálních zákonů. Lidské oko je v této disciplíně velmi obratné, a proto je v obecném rozsahu velmi složité dosahovat alespoň podobných výsledků, jaké dostává od tohoto orgánu lidský mozek. Ten má totiž k dispozici schopnost učení a rozsáhlou paměť zkušeností, které velmi dobře využívá ke kategorizaci obrazu. V některých úzkých oblastech dokáží metody analýzy obrazu přesto dosahovat kvalit lidského oka, a proto má smysl se jimi dále zabývat. Tyto oblasti jsou často vymezeny konkrétními vlastnostmi, na základě kterých lze zvolit efektivní metodu analýzy. Mezi tyto vlastnosti může patřit například barva nebo tvar objektu.

1.2 Porovnání fotografií

Častou disciplínou analýzy obrazu je porovnání dvou a více obrazů a získání informace o tom, jak moc jsou si dané obrazy podobné. Tato úloha se nazývá QBE (Query By Example), tedy hledání pomocí příkladu. Její odlišnost od ostatních úloh tkví v tom, že místo slovního popisu, jaké výsledky hledáme, máme k dispozici jeden konkrétní výsledek, odpovídající našim požadavkům. Hledáme tedy další podobné výsledky. To nám výrazně zvyšuje šance vyřešit tuto úlohu správně, protože místo několika podmínek popisujících zadání máme k dispozici velké množství obrazových dat, které můžeme k řešení využít.

2 Existující metody porovnávání obrazů

Jak jsem již zmínil, obrazy můžeme porovnávat podle mnoha různých kritérií, mezi které patří například barva nebo tvar. Na základě získaných informací poté stanovíme tzv. vektory vlastností (feature vectors), které můžeme navzájem porovnávat. Takový vektor reprezentuje daný obrázek tak, aby byl co možná nejkratší, ale zároveň obsahoval maximum důležitých informací o obraze. K získávání vektorů i k jejich porovnávání můžeme použít mnoho různých algoritmů. Uveďme si tedy několik nejpoužívanějších metod porovnání obrazu včetně jejich výhod a nevýhod.

2.1 Metody využívající barvu

Tyto metody pracují na principu histogramu, tedy určí četnost výskytu konkrétních barev, na základě kterých mohou porovnávat obrazy mezi sebou. Pokud tedy dva obrazy obsahují přibližně stejné množství zastoupení jednotlivých barevných složek, lze hovořit o jejich podobnosti. Problém však nastává s velikostí barevného spektra, kterým může být obraz reprezentován. Pokud je barva pro každý pixel reprezentována 32 bity, může pouhou změnou kontrastu dojít k neshodě dvou obrazů. Je tedy vhodné toto spektrum výrazně zmenšit.

Lepší výsledky totiž dostaneme v případě, kdy budeme porovnávat množství od sebe vzdálenějších barev v barevném spektru (třeba červenou a modrou) dvou obrazů, než množství dvou přímo sousedících barev (světle a tmavě červená). Jinými slovy to tedy znamená, že více podobné jsou si obrazy, které obsahují přibližně stejné množství červených a modrých pixelů než obrazy, které mají přibližně stejné množství světle červených a tmavě červených pixelů. Následující metody se tedy liší ve způsobu redukce množství porovnávaných barev. Detailní informace o jednotlivých metodách lze nalézt v kapitole 10 v knize J. R. Parkera [2].

Histogram

Histogram reprezentuje četnost výskytu určité barvy v konkrétním obraze. U spojitých obrazů lze histogram jednoduše získat jako sumu pozorování, z nichž každé patří do disjunktní množiny. Celkový počet pozorování označme jako n a k je celkový počet disjunktních množin. Nejen u spojitých obrazů můžeme histogram vnímat také jako pravděpodobnost výskytu konkrétní barvy, tedy náhodné proměnné v daném obraze. Nejčastěji je histogram reprezentován sloupcovým nebo spojitým grafem, kde na ose x je barva a na ose y je zobrazena četnost výskytu konkrétní barvy.

$$n = \sum_{i=1}^{k} m_i$$

Histogram se může vyskytovat ještě v kumulativní podobě, která hodnotu výskytu určité barvy přičte k hodnotě pro předchozí sousední barvu. Grafem je pak neustále rostoucí funkce.

2.1.1 Metoda střední barvy

Tato metoda spočívá v aproximaci obrazu do jedné skalární hodnoty. Nad celým obrazem tedy vypočítáme jedinou výslednou barvu. Tato barva je součtem hodnot všech pixelů pro každou složku s následným vydělením celkovým počtem pixelů v obrázku. Jedná se tedy o běžný aritmetický průměr.

Tato metoda je velmi jednoduchá a rychlá, protože porovnat jeden údaj je velmi snadné. Nedosahuje však příliš dobrých výsledků, protože při ní ztratíme příliš mnoho informací o původním obraze. Není tedy příliš vhodné popisovat obrazy jedinou skalární hodnotou.

2.1.2 Metoda Color Quad Tree

Metoda Color Quad Tree je založena na sestavení několika úrovňového grafu typu strom. Tento strom je tvořen postupně směrem od listů ke kořeni. Každý uzel je tvořen právě čtyřmi listy a každé čtyři uzly mohou tvořit právě jeden další uzel. Listy jsou v obraze zvoleny jako čtverec nad čtyřmi přímo sousedícími pixely. Počet uzlů v každé úrovni odpovídá vždy mocnině čtyř, podle toho, o kterou úroveň se jedná. Kořen bude tedy jeden, další úroveň bude obsahovat 4 uzly a další úroveň bude mít uzlů již 16 a tak dále.

Hodnota uzlu se z listů může spočítat několika způsoby. Ten nejjednodušší je opět průměr. Způsobů porovnání obrazů mezi sebou může být také více. Můžeme si například zvolit konkrétní úroveň stromu a porovnat tyto úrovně mezi sebou.

Tato metoda bude o něco přesnější než metoda střední barvy a její přesnost můžeme ovlivnit tím, v jaké hloubce budeme stromy srovnávat. Metoda však selže například pokud bude jeden z obrazů rotován o určitý úhel.

2.1.3 Histogram odstínu a intenzity

Vzhledem k tomu, že barva je tvořena vždy tří-složkovým vektorem R, G, B, je o něco složitější s nimi pracovat. Jednodušší variantou je tedy skalární hodnota, jako například odstín (šedi) nebo intenzita. Sestavit pak takový histogram je velmi jednoduché, a i lépe porovnatelné než histogramy jednotlivých barevných složek.

Porovnání dvou histogramů

Dva histogramy jsou shodné právě tehdy, když se četnost složky jednoho histogramu $H_1[i]$ rovná četnosti složky druhého histogramu $H_2[i]$ a to pro všechny složky.

Podobnost dvou histogramů se však určuje o něco komplikovaněji. Histogramy jsou N-složkové vektory, a proto je vhodné využívat ke zjištění jejich podobnosti Euklidovskou metriku.

$$d_e(H_1, H_2) = \sqrt{\sum_{i=1}^{n}(H_1(i) - H_2(i))^2}$$

Tato metrika určuje vzdálenost dvou vektorů od sebe a hodí se pro porovnání dvou histogramů. Čím je číslo větší, tím jsou od sebe dva histogramy vzdálenější a tedy méně si podobné.

Pokud však chápeme histogram jako vektor, nastává zde od reality jeden podstatný rozdíl. Tím je fakt, že jednotlivé složky vektoru reprezentují ortogonální dimenzi, a jsou tedy na sobě nezávislé. Naproti tomu v histogramu jsou sousední barvy podobné. Proto pokud použijeme pouze Euklidovskou metriku, připravujeme se o velmi podstatnou souvislost.

Lepším řešením je hodnotit podobnost histogramů shodných a rozdílných částí. Za části, které se shodují hodnocení navýšíme a za části, které se neshodují budeme hodnocení snižovat. Sestavíme tak pro každý histogram nový, který se nazývá kumulativní. Barevnou hloubku níže označíme jako k.

$$c_1 = a_1, c_n = a_{n-1} + a_n; n \in < 2, k >$$

Nad těmito histogramy pak opět provedeme Euklidovskou metriku pro zjištění podobnosti. K tomuto hodnocení je vhodné, aby byly oba histogramy normalizované, tedy hodnoty jednotlivých složek náležely stejnému intervalu, tedy $< 0, 1 >$.

Dalším způsobem porovnání dvou vektorů je korelace. Tato funkce vyjadřuje závislost mezi dvěma histogramy. Korelační koeficient určuje míru této závislosti a může nabývat hodnot z intervalu $< -1, 1 >$, přičemž -1 označuje absolutní neshodu, 0 vyjadřuje náhodnou závislost a 1 vyjadřuje shodu. Výpočet korelace pak probíhá následujícím vztahem, kde N odpovídá počtu složek vektoru.

$$C(H_1, H_2) = \frac{\sum_i H'_1(i) \cdot H'_2(i)}{\sqrt{\sum_i H'_1(i) \cdot H'_2(i)}}$$

$$H'_k(i) = H_k(i) - (1/N)(\sum_j H_k(j))$$

Metoda Chi-square posuzuje shodnost opačně než korelace, ale podobně jako Euklidovská vzdálenost. Menší hodnota tedy vyjadřuje větší shodu a 0 reprezentuje absolutní shodu. Principiálně je založena na rozdělení pravděpodobnosti, kde pomocí distribuční funkce určíme míru shodnosti obou vektorů.

$$\chi^2(H_1, H_2) = \sum_i \frac{(H_1(i) - H_2(i))^2}{H_1(i) + H_2(i)}$$

Více o porovnávání histogramu lze nalézt v knize Learning OpenCV [7].

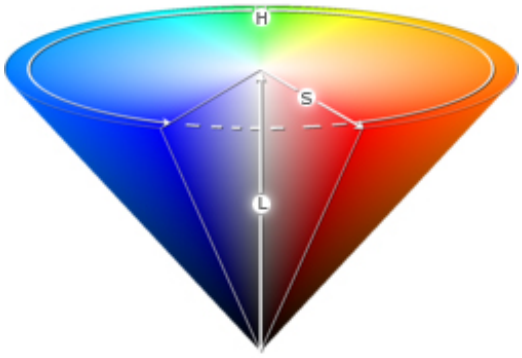
Barevné modely a jejich převod

Světlo je fyzikálně reprezentováno jako vlnění o určité vlnové délce. Při různých vlnových délkách vnímá lidské oko světlo jako různé barvy. Tato světelná záření se mohou skládat a tvořit tak mnoho variací barev. Na základě tohoto faktu vzniklo hned několik způsobů reprezentace barvy v diskrétním světě počítačů.

Tím nejznámějším a nejpoužívanějším barevným modelem je RGB. Z prvních písmen anglických názvů barev červené, zelené a modré lze snadno odvodit, jakým způsobem probíhá jejich skládání. Protože v tomto modelu vycházíme standardně z černé barvy a postupným přidáváním intenzity jednotlivých složek tvoříme konkrétní barvu, říkáme tomuto modelu aditivní. Tento model využíváme na většině displejů.

Naopak model CMY (cyan, magenta, yellow) nazýváme subtraktivním, protože vycházíme z barvy bílé a ubíráním intenzity se cílová barva postupně ztmavuje. Tento barevný model se využívá převážně k tisku, kde je základem skutečně bílý papír, což přináší značné výhody.

Oproti výše zmíněným modelům se HSL (hue, stauration, lightness) snaží více napodobit lidské vnímání barev. Jeho složky jsou tvořeny tónem, sytostí a jasem barvy. Velmi často se tato reprezentace zobrazuje ve trojrozměrném prostoru jako kužel, kdy tón barvy je zobrazen jako úhel na kruhové podstavě, sytost jako její poloměr a jas jako výška. Můžeme jej vidět na obrázku ze stránky `http://digital-intermediate.co.uk/colour/colourbasics.htm`, kde se také nachází mnoho doplňujících informací.



Obrázek 1: Reprezentace barevného spektra HSL pomocí kuželu

Převod RGB na model HSL lze pak spočítat dle následujících vztahů:

$$h = \begin{cases} \text{nedefinován}, & \text{jestliže } max = min \\ 60° \times \frac{g-b}{max-min} + 0°, & \text{jestliže } max = r \text{ a } g \geq b \\ 60° \times \frac{g-b}{max-min} + 360°, & \text{jestliže } max = r \text{ a } g < b \\ 60° \times \frac{b-r}{max-min} + 120°, & \text{jestliže } max = g \\ 60° \times \frac{r-g}{max-min} + 240°, & \text{jestliže } max = b \end{cases}$$

$$l = \frac{1}{2}(max + min)$$

$$s = \begin{cases} 0, & \text{jestliže } l = 0 \text{ nebo } max = min \\ \frac{max-min}{max+min} = \frac{max-min}{2l}, & \text{jestliže } 0 < l \leq \frac{1}{2} \\ \frac{max-min}{2-(max+min)} = \frac{max-min}{2-2l}, & \text{jestliže } l > \frac{1}{2} \end{cases}$$

2.1.4 Přepočítání barev

Většina obrazů je složena z mnoha různých barev. Tyto barvy jsou si často velmi podobné, a proto je vhodné tyto podobné barvy sloučit pod jedinou stejně jako v prvních dvou metodách. Stanovíme si však přesné barvy a rozsahy, kam kterou barvu budeme přepočítávat:

Barva	Hodnota
Red	$(170, 0, 0)$
Orange	$(170, 85, 0)$
Yellow	$(170, 0, 0)$
Green	$(0, 170, 0)$
Blue	$(0, 0, 170)$
Purple	$(85, 0, 170)$
Pea	$(85, 170, 0)$
Black	$(25, 25, 25)$
White	$(240, 240, 240)$
Grey	$(128, 128, 128)$

Tabulka 1: Tabulka barev

Tyto barvy mají mezi sebou stejné vzdálenosti, a proto je velmi jednoduché rozhodnout, pod kterou bude původní barva spadat. Po přepočítání tedy opět sestavíme histogram a použijeme Euklidovskou metriku.

Tato metoda je odolná vůči kontrastním změnám, a také nehraje roli rotace obrazu. Změna perspektivy, ze které byl například snímek objektu pořízen, bude činit této metodě již problémy.

2.2 Metody využívající tvar

Tyto metody jsou založené na rozpoznávání konkrétních objektů v obraze na základě jejich tvaru. Oproti metodám založeným na barvě mají mnohem větší složitost. Těmto metodám se proto budu věnovat jen krátce z důvodu nevhodnosti použití pro mou praktickou implementaci. Celý proces těchto metod je rozdělen do několika samostatných bloků, z nich každou lze realizovat mnoha různými způsoby, které jsou podrobně popsány doc. Sojkou [3].

2.2.1 Segmentace obrazu

V tomto kroku se využívají různé metody detekce hran k extrakci objektu a k jeho následnému odlišení od pozadí. Samotná detekce hranice využívá derivaci k nalezení extrémů v signálu obrazu. Tyto extrémy pak pomocí prahování označíme za hrany. Mezi nejznámější patří například Cannyho detektor [4].

2.2.2 Rozpoznávání příznaků

Cílem tohoto kroku je reprezentovat objekt v obraze pomocí vektoru tak, aby obsahoval všechny podstatné informace. Nalezení takovéhoto vektoru je velmi složité a nelze tento úkol vyřešit obecně. Vždy proto volíme specifický algoritmus podle toho, jaký typ objektu mají detekovat. Jako příklad můžeme uvést velikost plochy objektu nebo jeho křivost.

2.2.3 Klasifikace

Proces klasifikace řeší úlohu převedení vektoru na informaci, snadno zpracovatelnou pro člověka. V praxi to znamená například pojmenování objektu podstatným jménem (v obraze se nachází dům). Tento krok úzce závisí na předchozích dvou a jeho úspěšnost se odvíjí od kvality dat, které obdrží ke zpracování. V praxi se využívá například neuronových sítí a jejich procesu učení. Toto řešení se přibližuje tomu, jak stejnou úlohu zpracovává lidský mozek.

3 Vlastní řešení a praktická realizace

Pro účely svého řešení jsem se rozhodl využít již existujících metod porovnání obrazu z kapitoly 2 či jejich kombinaci. Díky tomu se mohu více věnovat jejich případné modifikaci na základě experimentu nebo optimalizace. Naopak při realizaci úlohy bych rád přinesl řešení nové, které mi umožní díky svým vlastnostem širší využití v budoucnu.

3.1 Již existující praktická řešení

Před návrhem svého řešení jsem nejdříve hledal, co je možné pro tuto úlohu využít a narazil jsem na velkou spoustu aplikací zabývajících se tímto tématem. Tyto programy jsem zařadil do dvou skupin, podle principu jejich fungování, a z každé této skupiny uvedu nějaký příklad. Na těchto webových stránkách si lze prohlédnout několik již existujících aplikací.

- `http://www.google.com/imghp`
- `http://www.duplicate-finder.com`
- `http://www.bing.com/images/`
- `http://www.visipics.info`
- `http://www.tineye.com`
- `http://labs.systemone.at/retrievr/`
- `http://duplifinder.codeplex.com`

3.1.1 Offline zpracování na lokálním PC

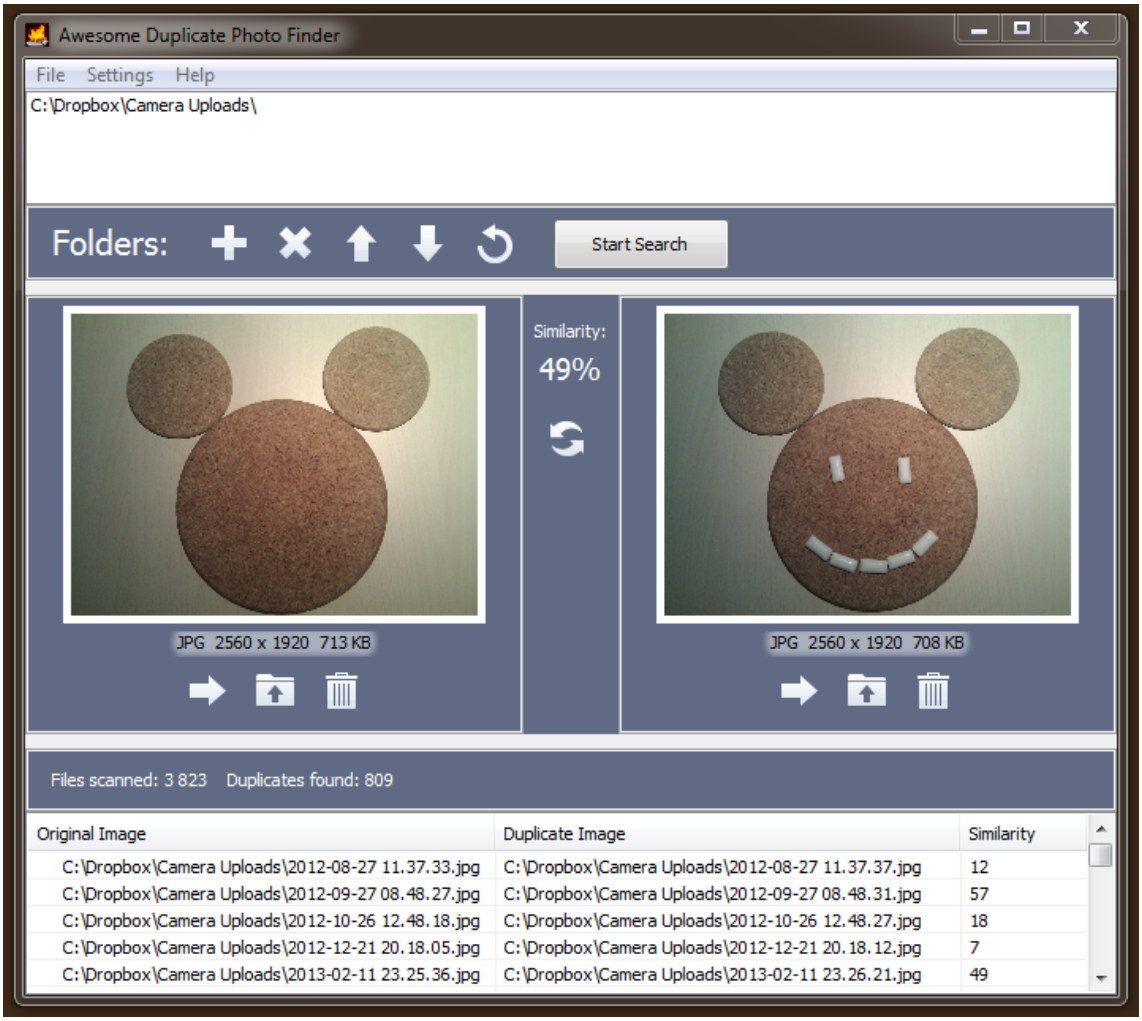
Tento způsob řešení velmi dobře demonstruje jednoduchá aplikace Awesome Duplicate Photo Finder, která si klade za cíl nalézt duplicity v lokální sbírce fotek pro ušetření prostoru na disku. Pracuje velmi spolehlivě a je velmi jednoduše ovladatelná. Praktické výsledky sice nejsou vždy oslňující, ale svůj hlavní účel plní docela dobře.

Výhody:

- Fotky není třeba nahrávat na vzdálený server z čehož vyplývá vyšší bezpečnost dat.
- Žádné nároky na propustnost linky.

Nevýhody:

- Limitace výkonu lokálního počítače.
- Možnost pracovat pouze s fotkami, které se nachází na lokálním disku.
- Aplikace je dostupná pouze pro Windows. Případná portace na další platformy vyžaduje nemalou režii.



Obrázek 2: Ukázka programu Awesome Duplicate Photo Finder

3.1.2 Online řešení

V této skupině je jednoznačným příkladem Google Images (`https://www.google.cz/imghp`), který umožňuje hledat obrázky nejen na základě vzoru napříč internetem.

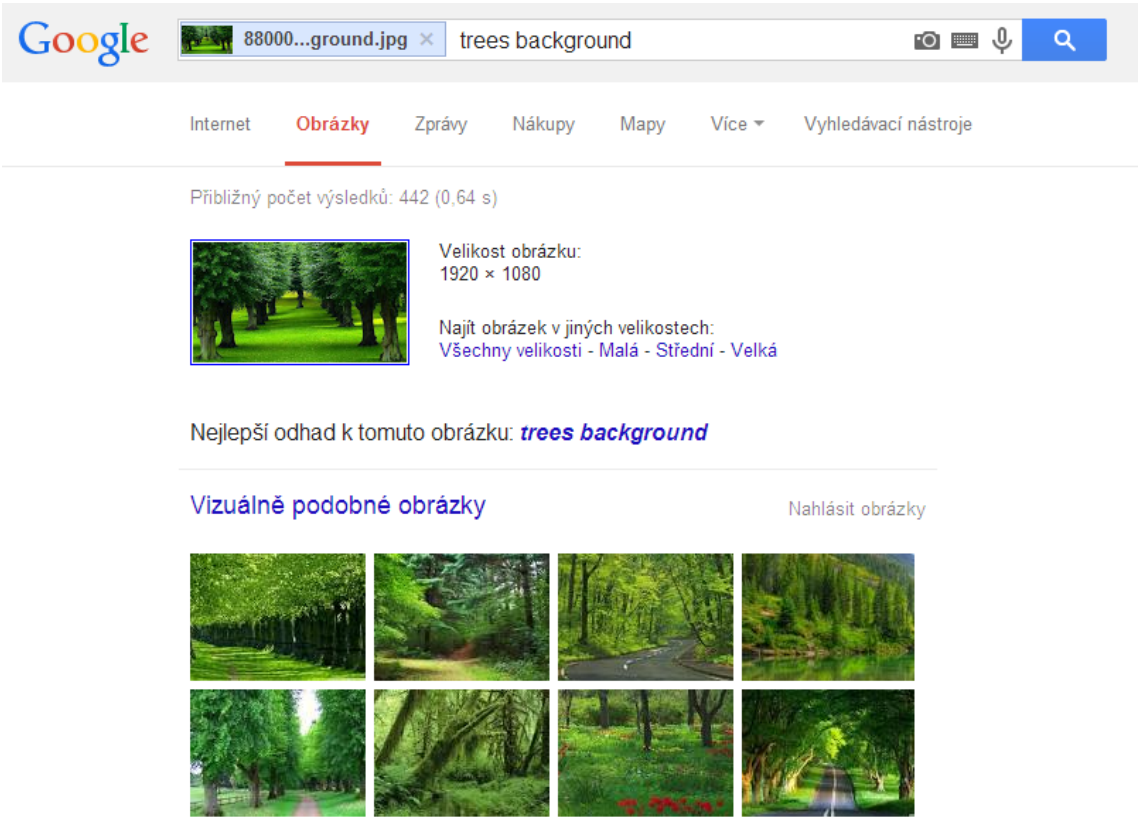
Výhody:

- Obrovská databáze obrázků.
- Vysoký výpočetní výkon datového centra.
- Možnost navázání na další funkce (vyhledání stránek, sdílení).
- Minimální nároky na výkon lokálního počítače.

- Aplikaci lze využít na libovolném zařízení (desktop, mobil, tablet) a operačním systému (Linux, Windows, Android) s internetovým prohlížečem.

Nevýhody:

- Nutnost nahrát fotografii na vzdálený server, a tedy horší bezpečnost dat.
- Vysoké nároky na propustnost linky vyžadují nemalou režii.



Obrázek 3: Ukázka Google Images

3.2 Stanovení požadavků vlastního řešení

Ná základě zkušeností s již dostupnými řešeními jsem si stanovil své cíle, které od svého řešení vyžaduji. Tyto cíle kombinují vlastnosti obou výše zmíněných skupin, a proto si dovolím svůj teoretický návrh řešení nazvat hybridním. Kombinací vlastností Offline i Online přístupu si kladu za cíl eliminovat většinu nevýhod obou řešení a získat případně výhody nové.

3.2.1 Webová aplikace

V dnešní době je internetový prohlížeč s podporou HTML5 téměř v každém zařízení, a proto je volba této sady technologií na místě. Další nespornou výhodou je proces "instalace"takové aplikace, kdy stačí jen zadat příslušnou adresu do webového prohlížeče. Na rozdíl od běžných desktopových či mobilních aplikací je doba trvání procesu prvního spuštění o mnoho kratší. Dlouhodobá udržitelnost aplikace a rozšiřování funkčnosti v průběhu času je také velmi jednoduchá. Nemůže totiž nastat situace, kdy různí klienti mají různé verze, a proto není třeba brát při vývoji ohled na starší verze své vlastní aplikace či proces updatování.

Díky čím dál výkonnějším mobilním zařízením je vhodné cílit i na tuto neustále se zvětšující skupinu uživatelů. Rovněž je velmi praktické využít CSS3 Media Queries pro přizpůsobení aplikace dané velikosti a rozlišení obrazovky pro pohodlnější ovládání. Tato přenosná zařízení mají dnes již velmi kvalitní fotoaparát, a proto na nich lze ovládání zjednodušit i tím, že jim umožním fotky pomocí vestavěného fotoaparátu rovnou pořizovat. Tímto se pro uživatele mobilních zařízení otevírá zcela rozdílný způsob využití.

3.2.2 Tlustý klient

Tento přístup jsem si vybral jednak s rostoucím zájmem o takovýto druh aplikace, a také kvůli rychlosti odezvy samotné aplikace pro uživatele. Bude tedy stačit stáhnout aplikaci jednou a po celou dobu jejího používání bude se serverem komunikovat pouze v podobě zpráv při výměně dat. Server poslouží pouze jako databáze obrázků a úložiště pro klientskou část aplikace.

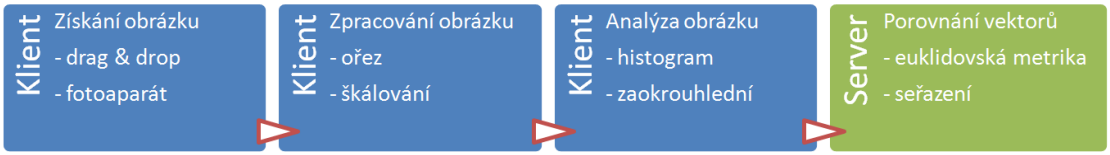
Z tohoto důvodu jsem se rozhodl provádět celý proces analýzy obrazu již u klienta a server vždy požádat pouze o samotné porovnání obrázků mezi sebou. To znamená, že nebude nutné nahrávat na server pokaždé daný obrázek, ale pouze vektor s informacemi o obrázku, což vede k dramatické úspoře dat při komunikaci. Další nespornou výhodou je bezpečnost. Pokud samotný obrázek neopustí klientský počítač, je velmi dobře možno garantovat zabezpečení aplikace. U obrázků, které jsou například citlivým duševním majetkem, je tento parametr velmi důležitý.

Tento přístup mi sice nedovolí využít ke zpracování tak velkého výkonu, jako například v případě Google, ale bude jej naopak možno velmi dobře používat na mobilních zařízeních s pomalým připojením k internetu. Dále bude možné obsloužit obrovské množství uživatelů s relativně malým výkonem serveru i klienta, z čehož v praxi plynou nízké náklady na provoz takovéto aplikace. Zároveň však nepřijdu o možnost propojování dat s dalšími informacemi, a mohu tak své řešení rozšířit o velké množství dalších funkcí.

3.3 Návrh samotného algoritmu analýzy obrazu

Vzhledem ke stanoveným požadavkům řešení je nutné brát ohled na výkon potřebný ke zpracování. Nelze tedy využít metod postavených na tvaru objektu a detekci hran. Naopak je vhodné využít barvu jako klíčový parametr obrazu. Algoritmy založené na barevném histogramu mi umožní rychlý běh aplikace a zároveň mi ještě dovolí poskytovat

uživateli poměrně dobré výsledky. Je to tedy vzhledem k omezeným možnostem výkonu nejoptimálnější varianta.



Obrázek 4: Úlohy, které zpracovává klient a server.

3.3.1 Zpracování obrazu

Prvním krokem před samotnou analýzou je zpracování obrazu. Díky tomu je možno výrazně zlepšit výsledky procesu analýzy. V mém případě jsem se rozhodl z důvodu omezeného výkonu pouze oříznout obraz na největší vnitřní čtverec, abych zajistil vždy stejný vstup pro analyzátor. Pro čtverec jsem se rozhodl z důvodu různé orientace vstupních fotografií. Pokud bych použil například obdélník, který je orientován na šířku, byl bych nucen fotografiím orientovaných na výšku oříznout mnohem větší část než těm orientovaným na šířku. Tím bych je samozřejmě znevýhodnil. Čtverec je tedy nejoptimálnější volba pro oba případy orientace.

Další krok, který jsem se rozhodl použít je škálování. Přepočítáním všech obrázků na jednotné rozlišení analyzátoru umožním vyhodnotit lepší výsledky, což výrazně ovlivní nejen rychlost samotné analýzy. Další pozitivní dopad má tento proces na množství spotřebované RAM paměti, což je především kritické pro mobilní zařízení.

3.3.2 Analýza obrazu

Z důvodu požadavku na rychlost jsem se rozhodl využít kombinaci více metod z kapitoly 2. Princip mého algoritmu je založen primárně na vyhodnocení histogramu v obraze a následné průměrování jednotlivých barev z důvodu zpřesnění výsledků a minimalizace přenášených dat. Při počítání výskytu četnosti barev využívám pouze klasický histogram opět z důvodů výkonu. Samotnému průměrování se budu věnovat níže, v kapitole experimentů 4.

3.3.3 Porovnávání obrazů

Tato část výpočtu se jako jediná provádí na serveru, a díky tomu lze výslednou aplikaci velmi dobře škálovat. V kapitole 2 uvádím několik způsobů řešení této úlohy. Vybral jsem si tedy výše zmíněnou Euklidovskou vzdálenost, se kterou mám již dřívější dobré praktické zkušenosti. Z důvodu výkonu nebudu používat kumulativní histogram.

3.4 Zvolené technologie

Protože existuje obrovské množství různých způsobu technického řešení a návrhu architektury mé aplikace, spoléhal jsem se při rozhodování především na praktické zkušenosti s jednotlivými technologiemi a frameworky. Snažil jsem se tedy jednotlivé přístupy a technologie vyzkoušet na krátkých a jednoduchých příkladech a seznámit se tak alespoň ve zkratce s daným frameworkem.

Velmi mne při tom inspiroval Addy Osmani a jeho kniha Learning JavaScript Design Patterns [5]. Proto mi při hledání optimální architektury aplikace výborně posloužil projekt s názvem TodoMVC (`http://todomvc.com`) od stejného autora. Zde je k dispozici jednoduchá aplikace pro správu úkolů opakovaně navržená pomocí obrovského množství různých JavaScriptových frameworků s mnoha různými architekturami a přístupy. Díky tomu jsem se mohl velmi blízce seznámit se skutečnými výhodami i nevýhodami, které každé řešení nabízí.Po dlouhém zvažování jsem se nakonec rozhodl zvolit následující sadu technologií a přístupů.

3.4.1 Serverová část aplikace

U této části aplikace není v mém řešení na prvním místě výkon či použitá platforma, ale především spolehlivost samotné infrastruktury a škálovatelnost v případě větší zátěže. Z tohoto důvodu jsem zcela upustil od klasického webhostingu nebo virtuálního či dedikovaného serveru, ale vsadil jsem na tzv. PaaS řešení, které mi nabízí vše, co pro svou aplikaci požaduji.

Platform as a Service (PaaS)

Jedná se o výpočetní platformu z kategorie Cloud Computing, která poskytuje prostředí pro běh aplikace. Mezi hlavní výhody patří abstrakce od síťové, hardwarové a softwarové vrstvy zajišťující provoz. Není se tedy třeba starat o údržbu a výměnu hardwaru nebo aktualizace operačního systému a ostatního podpůrného softwaru. Další výhodou je škálovatelnost celého řešení. Díky virtualizaci a load balancingu je mé aplikaci přiděleno vždy tolik prostředků, kolik právě potřebuje napříč fyzickými servery, a proto se není třeba obávat výpadku při zvýšené zátěži. Další nespornou výhodou je replikace mezi různými datovými centry rozmístěnými na geograficky rozdílných místech světa. To umožňuje zkrátit dobu odezvy, díky využití vždy nejbližšího možného datového centra k místu, kde se uživatel zrovna nachází. Další výhodou je také zálohování dat, které bývá často opomíjeno.

Výběr a použití cloudové platformy

Mezi nejznámější poskytovatele PaaS služeb patří například Microsoft Azure, Amazon Web Services nebo Google App Engine. Rozhodl jsem se pro Google App Engine především z důvodu jednoduché integrace autentifikace uživatelů pomocí Google účtu. Tento aspekt je pro mne důležitý především kvůli uživatelům, kteří přistupují k mé aplikaci z chytrých mobilních telefonů a tabletů. Dle aktuálně dostupných statistik podílu

jednotlivých mobilních operačních systémů na trhu je nejrozšířenější Google Android, a to jak pro Českou Republiku, tak i celosvětově. Právě díky provázanosti služeb Googlu je tedy autentifikace uživatelů na operačním systému Android mnohem jednodušší a pohodlnější.

K ukládání dat na serveru je pro mne velmi důležitá databáze obrázků a vektorů, nad kterými budu potřebovat vyhledávat. Tento problém velmi dobře řeší aplikační rozhraní, která jsou na App Enginu k dispozici. Je to DataStore, pro ukládání textových a numerických dat, a BlobStore, pro ukládání binárních souborů. Obě tyto uložiště jsou velmi dobře provázaná a dovolují mi například velmi dobře pracovat s obrázky. Mnoho důležitých informací o těchto rozhraních jsem se dočetl v oficiální dokumentaci a v knize Programming Google App Engine [8].

Jako programovací jazyk jsem si zvolil Python, především kvůli své jednoduchosti a rychlosti vývoje. Mým kritériem zde bylo dynamické typování. Velmi dobře se mi totiž pracuje, když mohu na serveru i klientu používat podobný přístup při psaní kódu, což mi dynamická typovost na obou stranách umožňuje. Na serverové části není kromě vyhledávání nutná žádná složitá aplikační logika, a proto jsem neuvažoval o nasazení jazyka Ruby. Na App Enginu je pro Python k dispozici jednoduchý webový framework webapp2, který zpřístupňuje jednotlivá aplikační rozhraní do mé aplikace a ulehčuje práci s nimi.

3.4.2 Klientská část aplikace

Jak jsem již zmínil, k realizaci klientské části jsem se rozhodl pro HTML5 na rozdíl od Javy nebo Flashe. Je to také jediný způsob, jak v dnešní době tvořit multiplatformní aplikaci pokrývající také mobilní zařízení, bez nutnosti instalace dalšího podpůrného plug-inu či jiné aplikace. Při výběru JavaScriptového frameworku jsem se rozhodoval především na základě šablonovacího systému, rychlosti běhu výsledné aplikace a nutného množství přenesených dat ke klientovi.

Nejprve jsem uvažoval nad frameworkem AngularJS, který se v současné době těší velké oblibě nejen díky svému propracovanému "two way data binding". Později jsem se ale rozhodl využít celou sadu nástrojů Google Closure Tools. Tyto nástroje jsou rozděleny do tří hlavních celků.

Google Closure Tools

Prvním je zmíněný šablonovací systém s názvem Closure Templates. Ten má oproti ostatním obrovskou výhodu především v rychlosti. Šablony lze totiž dopředu zkompilovat a uživateli zasílat již jen optimalizovanou verzi. Tyto šablony jsou zkompilovány do jednoduchých metod, které stačí pouze volat, a tak velmi rychle upravovat *document object model* (DOM) [5] samotné stránky. Další výhodou těchto šablon je automatické escapování. To zajišťuje zabezpečení aplikace proti útokům Cross Site Scripting.

Druhou částí je Closure Library. Jedná se o knihovnu již hotových řešení opakujících se problémů při vývoji. Výhodou této knihovny je především její spolehlivost a dokumentační komentáře přímo uvnitř kódu. Díky obrovskému pokrytí jednotlivých částí knihovny

Unit Testy je možné se na ni spoléhat a díky dokumentačním komentářům jsem se naučil mnoho nového o samotném JavaScriptu, ale také o způsobu přemýšlení tvůrců knihovny.

Mezi nejpraktičtější nástroje této knihovny patří například systém správy závislostí. Ten umožňuje kód rozdělit do mnoha menších celků, které mají mezi sebou pomocí $goog.require()$ nadefinované závislosti, což přispívá k lepší organizaci kódu. Další zajímavou funkcí je podpora jmenných prostor, i když je samotný JavaScript nepodporuje. Jako poslední bych zmínil ještě pokrytí inkonzistence mezi jednotlivými prohlížeči, a to i na úrovni rozdílných verzí stejného prohlížeče. Díky tomu je zajištěna stabilita aplikace a nemusím se obávat, že by se chovala v různých prohlížečích jinak.

Posledním a nejdůležitějším je Closure Compiler. Je to aplikace, které předáme zdrojový kód v JavaScriptu a výstupem je optimalizovaná verze. Mezi nejzajímavější funkce tohoto kompilátoru patří kompilace šablon, o kterých jsem se již zmiňoval výše. Dále se stará o statickou analýzu kódu, což je záležitost, kterou jinak dělá až prohlížeč u uživatele. Díky tomu lze odhalit spoustu chyb ještě před tím, než dojde k nasazení (deployment) aplikace do ostrého provozu. Kompilátor se také stará o minifikaci a obfuskaci, což mu pomáhá zdokonalit volitelné typování pomocí dokumentačních komentářů. Za zmínku stojí také odstraňování nepoužívaného kódu. V praxi to tedy znamená, že mohu mít například třídu s mnoha metodami, ale pokud volám jen jedinou, kompilátor ostatní metody odstraní, a zmenší tak objem výsledné aplikace.

Transpilery

Dále jsem se rozhodl psát celou aplikaci v CoffeeScriptu, což je jazyk, který lze transpilovat do JavaScriptu. Transpiler na vstupu přečte zdrojový kód jazyka CoffeeScript a na výstup vygeneruje zdrojový kód jazyka JavaScript v optimální a zabezpečené verzi. Přináší totiž mnoho výhod, mezi které patří například zápis tříd pomocí klíčového slova $class$ nebo zjednodušený zápis a použití cyklů. Největší benefit však vidím v jeho syntaxi, která se mnohem více přibližuje Pythonu, než původní JavaScript. Díky tomu mohu mnohem rychleji přecházet mezi serverovou a klientskou částí aplikace a provádět potřebné úpravy.

Neméně podstatnými jsou transpilery kaskádových stylů. Z nich využívám Stylus a LESS. Díky jejich rozšíření CSS je velice snadné používat například proměnné, mixiny (opakující se části kódu podobné běžným funkcím) nebo přehlednější zanořování. Kód lze tedy mnohem lépe organizovat a tvořit.

3.4.3 Nástroje

Volba správných nástrojů je velmi důležitá při vývoji každé aplikace. Tyto nástroje mohou dramaticky ovlivnit průběh vývoje celé aplikace. Nástroje pro zautomatizování rutinní práce dokáží zvýšit produktivitu a soustředění programátora. Balíčkovací systémy zjednodušují údržbu kódů aplikace a umožňují velmi snadné rozšiřování funkčnosti.

Správa externích závislostí

Při vývoji jsem narazil na nepříjemný problém, který se týkal knihoven a nástrojů třetích stran. Po dobu vývoje mé aplikace několikrát vyšla novější verze například Google Closure Library, která opravovala nějakou chybu či doplňovala novou funkčnost. Takový proces ode mne vyžadoval neustále sledovat, zda vývojáři nevydali novou verzi a pokud ano, stáhnul jsem ji a nahradil v mé aplikaci. Tento proces mne velmi zdržoval, a proto jsem se rozhodl využít balíčkovací systémy NPM a Bower, aby tuto nepříjemnou práci udělaly za mne.

NPM je výchozí balíčkovací systém platformy Node.JS. Přestože na serverové části využívám Python a nikoli Node.JS, používám tento balíčkovací systém ke správě závislostí samotného Boweru a také nástroje Grunt, o kterém se zmíním později. Prakticky to funguje tak, že existuje centrální repozitář `http://npmjs.org`, kde se nachází informace o jednotlivých balíčcích. V mé aplikaci pak už jen vytvořím soubor "package"ve formátu JSON, kam uvedu všechny závislosti, které má aplikace vyžaduje. Díky sémantickému verzování je také zajištěna minimalizace chyb se zpětnou kompatibilitou, což výrazně usnadňuje a zpřehledňuje práci s balíčky. Pomocí příkazu v konzoli "npm install"pak stačí už jen stáhnout všechny potřebné závislosti a lze je začít používat. Jak můžete tušit, příkaz "npm update"vyřeší výše zmiňovanou situaci zcela automaticky během několika vteřin.

Na rozdíl od NPM se Bower stará o externí závislosti pro balíčky související s klientskou částí aplikace. Ke konfiguraci se obdobně používá soubor "bower"také ve formátu JSON. Bower však nevyužívá jeden centrální repozitář, ale je třeba u každého balíčku uvést cestu k samotnému repozitáři balíčku, nejčastěji tedy GIT. Opět lze využít sémantického verzování a samotná správa závislostí se ovládá také obdobně, tedy "bower install"a "bower update". Tento balíčkovací systém používám například pro Google Closure Tools nebo pro CSS framework Bootstrap. Za zmínku ještě stojí fakt, že tento nástroj vznikl ve společnosti Twitter.

Automatické spouštění úloh

Díky automatické správě externích závislostí jsem si výrazně zjednodušil část práce. Díky transpilaci CoffeeScriptu, LESS, Stylus a kompilaci pomocí Closure Compileru mi ale přibylo trochu více režie při vývoji a ladění. Z tohoto důvodu jsem se rozhodl tyto kompilace zautomatizovat také. Výborně mi k tomu posloužil nástroj Grunt.

Grunt je spoustěč automatických úloh. Pomocí souboru "Gruntfile"ve formátu js nebo coffee mu lze nadefinovat jednotlivé úkoly a využít k tomu spoustu externích modulů. Příkazem "grunt"tedy spustím své úlohy, které při spuštění provedou transpilaci CSS a CoffeeScriptu, kompilaci pomocí Closure Compileru, spustění Unit testů a nakonec spuštění samotného testovacího serveru App Enginu. Některé moduly mohou například sledovat konkrétní adresář a při každé změně spustit transpilaci. Tímto způsobem jsem si nadefinoval pomocí různých modulů svou rutinu, která se spustí pokaždé, když uložím soubor v adresáři s aplikací. Díky tomu mohu ve vedlejším okně sledovat, zda nemám ve svém kódu nějakou chybu. Pokud se nějaká chyba vyskytne, jsem na to upozorněn zvukovým signálem a hlášením o chybě zvýrazněným červenou barvou. Takto mohu

přijít na chyby nejen syntaktické, ale díky statické analýze Closure Compileru například i neshodujících se datových typů a podobně. To vše, aniž bych musel obnovit stránku v prohlížeči.

Tou nejpraktičtější věcí je však takzvaný LiveRealod, který se v poslední době velmi rozšířil v komunitě webových vývojářů. Prakticky spočívá v tom, že při prvním načtení stránky se spustí jednoduchý skript, který pomocí WebSockets poslouchá na portu 35 729. Pokud Grunt detekuje změnu například JS souboru, odešle tuto informaci webové stránce, která se automaticky obnoví s upraveným skriptem. Pokud se však jedná o CSS soubor, Grunt opět odešle tuto informaci stránce, ale ta tentokrát obnoví pouze CSS soubor. Díky tomu se mohu plně soustředit na tvorbu kódu a pouze sledovat výsledek svých změn v konzoli nebo přímo v prohlížeči jen krátkou chvíli poté, co změny provedu. Tento postup výrazně zrychluje práci, a také napomáhá lepšímu soustředění.

Pomocí Gruntu lze nastavit spouštění rozdílných úkolů pro rozdílné prostředí aplikace. Pro vývoj mám tedy nastavenou jinou sadu úkolů než pro nasazení na produkční server. Na rozdíl od vývoje nastavuji přísnější pravidla pro kontrolu a minifikaci především Closure Compileru, kterému se daří celou aplikaci výrazně zmenšit. To zajišťuje dobrý výkon a odezvy aplikace i na pomalejších mobilních zařízeních.

3.5 Realizace

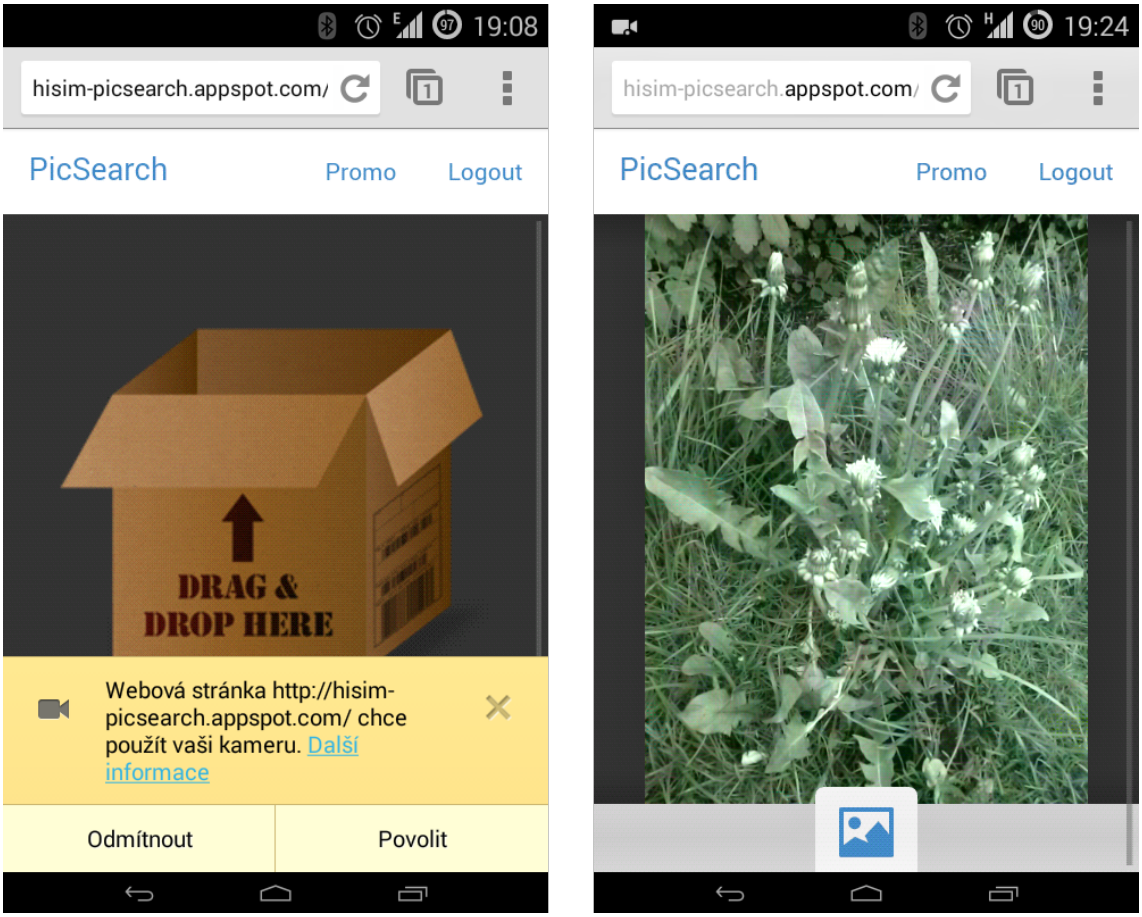
Díky důkladné analýze celého problému a nastavením dobrého pracovního procesu probíhal vývoj velmi rychle bez žádných větších problémů. Aplikaci jsem rozdělil do dvou hlavních částí. První část je určena pro nepřihlášené uživatele a má být především informativní. Pro přechod do funkční části se musí uživatel autentifikovat pomocí Google účtu.

Ověření uživatelů jsem zavedl z důvodu možného rozšíření. V tuto chvíli se s uživatelským účtem spojují pouze fotky, které do aplikace nahrál a lze s nimi případně v budoucnu dále pracovat. Google účet jsem zvolil především kvůli jednoduchosti přihlášení na mobilních zařízeních s operačním systémem Android, a také pro odstínění se od citlivých uživatelských dat. Ta totiž spravuje pouze Google a pro mne je uživatel reprezentován pouze unikátním identifikátorem, což mi pro tyto účely zcela postačuje.

3.5.1 Ovládání a chování aplikace

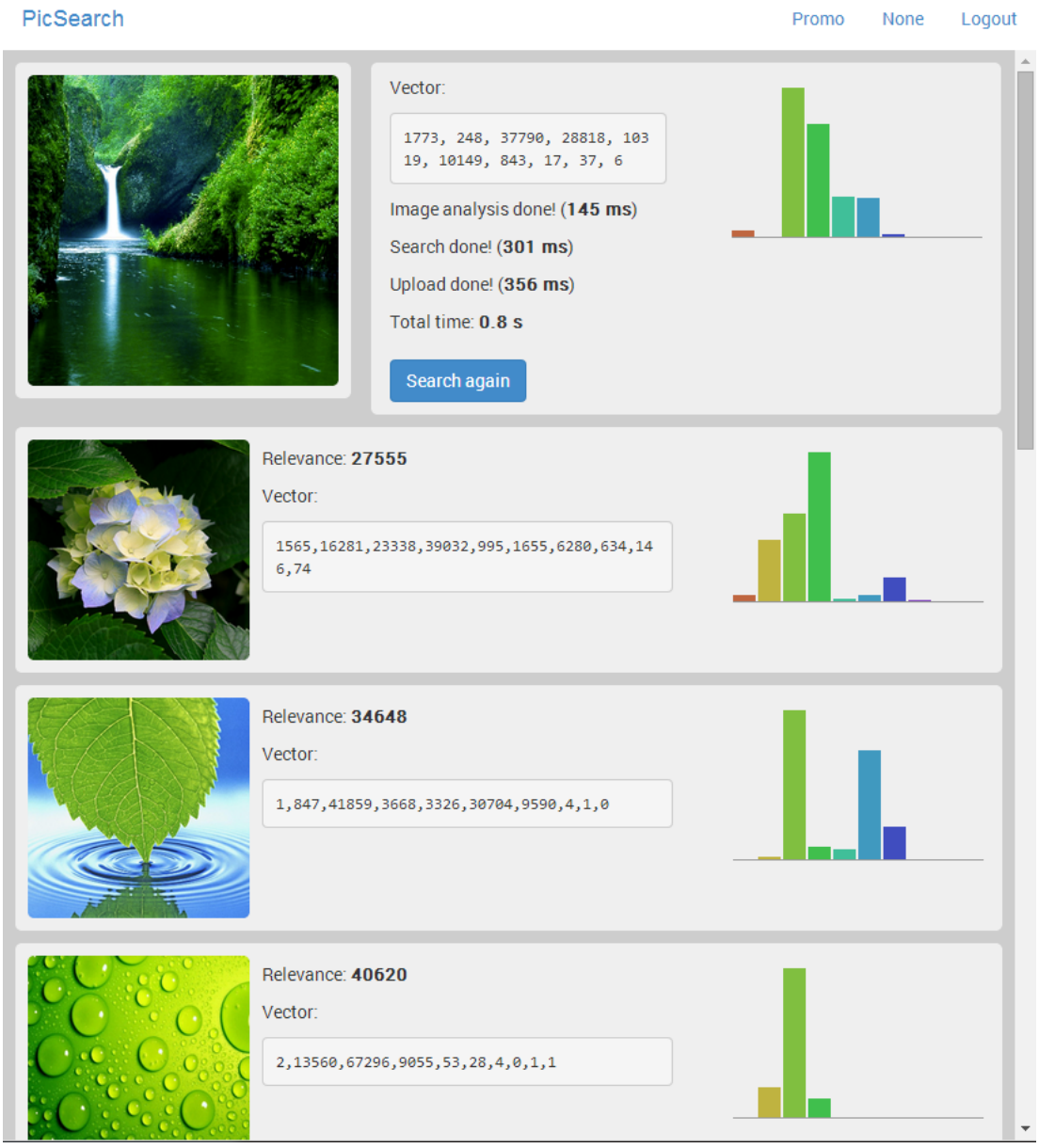
Ovládání aplikace se drobně liší pro uživatele klasického počítače a mobilního zařízení. Mobilní uživatelé jsou požádání webovým prohlížečem o povolení přístupu k fotoaparátu v jejich zařízení a ihned, jak jej potvrdí, se většina obrazovky zaplní živým náhledem toho, kam je fotoaparát namířen. Po kliku na náhled nebo na připravené tlačítko se aktuální fotka začne zpracovávat a uživateli se zobrazí s náhledem této fotky i informace o průběhu a stavu daného dotazu. Po zpracování se na server odešle zmíněný vektor a jako odpověď uživatel obdrží výsledky. Teprve až po zobrazení výsledků uživateli je fotka odeslána na server. Toto nahrávání však nijak nezatěžuje vyhledávání a implementoval sem jej z důvodu případné tvorby historie vyhledávání, a také k rychlému rozšiřování databáze obrázků pro účely testování. Z pohledu reálného použití je však tento krok zcela volitelný.

Pro uživatele používající klasický počítač je k dispozici také kromě použití zmíněné kamerky navíc možnost přetáhnout libovolný obrázek myší do okna prohlížeče, tedy Drag and Drop. Po uvolnění obrázku se automaticky spustí vyhledávání stejným způsobem, jako uživatelům mobilním. Náhledy jsou však větší a prostor je využit optimálněji v závislosti na velikosti okna prohlížeče.



Obrázek 5: Použití vestavěné kamery mobilního zařízení

Oběma typům uživatelů je vedle výsledků zobrazen ještě histogram a číselné hodnoty vypočítaných vektorů za účelem ladění. Nepředpokládám, že by bylo vhodné tyto informace zobrazovat reálným uživatelům. Místo nich by bylo vhodnější nabídnout uživateli jiné relevantní údaje tak, jak to dělá například Google. Za účelem ukázky mé práce jsem je ale nechal schválně viditelné.



Obrázek 6: Ukázka výsledků vyhledávání

3.5.2 Technické provedení

Pro svou aplikaci jsem na serveru i klientovi použil architekturu MVC, se kterou mi pomohly dvě jednoduché knihovny. Na serveru jsem použil Google App Engine Boilerplate, která řeší především ověřování uživatelů a pro CoffeeScript jsem použil Este.JS od českého autora Daniela Steigerwalda. Tyto knihovny mi pouze pomohly odstínit se od neustále opakujících se problémů při tvorbě webových aplikací, proto se jimi nebudu dále zabývat a popíšu raději řešení samotného algoritmu.

Klientskou část aplikace jsem rozdělil do několika logických celků, kterým odpovídají jednotlivé třídy ve zdrojovém kódu. Jsou to Camera, Image, Convolution, Analysis a SearchDB. Každá třída má zodpovědnost pouze za úzkou skupinu úkolů, a díky tomu lze aplikaci velmi pohodlně udržovat a případně rozšiřovat. Nejzajímavější je samozřejmě třída Analysis, a proto se jí budu věnovat trochu podrobněji.

Třída Analysis nemá definované žádné závislosti na dalších třídách, proto dle správného použití zásad Dependency Injection není nutné použít konstruktor. Samotná inicializace výpočtů je však přeci jen nutná. Z tohoto důvodu se metoda $init$ stará o převzetí a zpracování dat k výpočtu a konfiguraci algoritmu průměrování barev. Samotný výpočet histogramu, jak již název napovídá, provádí metoda $solveHistogram$. Tyto dvě metody tvoří celé vnější rozhraní třídy, a proto je její použití velmi snadné a intuitivní.

```
###*
  @public
  @param canvasOriginal {Object}
  @param colorSegmentCount {number}
###
init: (canvasOriginal, colorSegmentCount = 5) ->
  @canvasOriginal = canvasOriginal
  @colorSegmentCount = colorSegmentCount

  @width = @canvasOriginal.width
  @height = @canvasOriginal.height

  srcCtx = @canvasOriginal.getContext('2d')

  @imageData = srcCtx.getImageData(0, 0, @width, @height)
```

Obrázek 7: Metoda init třídy Analysis

Analysis obsahuje také několik privátních metod, které řeší některé dílčí úkoly. Konkrétně se jedná o metodu rgbToHsl, která přijímá tři parametry (R, G, B) a vrací pole opět tří hodnot (H, S, L). Úkol této metody spočívá v převodu barvy reprezentované hodnotami jasu červené, zelené a modré do reprezentace HSL, tedy na tón barvy, sytost a hodnotu jasu. Díky tónu lze barvy seřadit do lineárního barevného spektra a to dále později průměrovat.

Přesto, že tato metoda spočítá všechny tři složky nové reprezentace, využívám pouze složku tónu (H). Tato složka totiž obsahuje dostatek informací k tomu, abych byl schopen odlišit barvy od sebe. I když se tedy lidskému oku zdá, že vidí dvě různé barvy, mohou se od sebe lišit pouze sytostí a úrovní jasu, ale tón mají stále stejný. To mi umožňuje již zde částečně provádět průměrování, protože při výpočtu histogramu poté seskupím rovnou barvy se stejným tónem. Také díky Closure Compileru mohu tuto metodu kvůli lepší čitelnosti nechat počítat všechny tři hodnoty, ale na produkčním serveru bude kód již automaticky upravený pouze pro výpočet jediné hodnoty tónu. Tím se tento kód stává velmi dobře znovupoužitelným a kompilátor jej optimalizuje vždy specificky pro danou aplikaci.

Výpočet histogramu je pak již triviální operace, kdy cyklicky projdu obrázek po jednotlivých pixelech a dle hodnoty tónu každé barvy do připraveného pole navýším hodnotu četnosti o jedničku. Při tomto výpočtu četnosti z důvodu lepšího výkonu rovnou provádím průměrování do předem zvoleného počtu výsledných hodnot pomocí prahování. Před výstupem již pouze histogram seřadím tak, aby odpovídal barevnému spektru HSL a tím je celý výpočet hotov.

Výslednou aplikaci jsem uveřejnil pomocí deployovacího nástroje AppEnginu na adresu `http://hisim-picsearch.appspot.com`. Zde je možné si aplikaci vyzkoušet a přesvědčit se o funkčnosti a popisovaných parametrech.

4 Experimenty

Při implementaci jsem se věnoval především tomu, abych zajistil správnou funkčnost celé aplikace a dodržel přehlednost kódu. Často jsem upřednostňoval jasnější a obsáhlejší řešení před rychlostí a optimálností. Tento postup mi pomohl spolehlivě a rychle vyvinout celou aplikaci a zároveň jsem získal mnoho prostoru pro experimentování a optimalizace. Většinu optimalizace na úrovni kódu totiž za mne provedl Closure Compiler.

4.1 Hledání optimálního počtu složek vektoru

Popis obrázku vektorem s co možná nejmenším počtem složek tak, abych nepřišel o důležité informace, zpětně hodnotím jako velmi těžkou úlohu, kterou neumím jednoznačně vyřešit obecně. Takto usuzuji, protože jsem vždy narazil při testování na nějakou skupinu obrázků, kterým aktuálně nastavený počet složek nevyhovoval. Další problém, na který jsem narazil, je subjektivní vnímání samotné podobností dvou obrázků. Následující závěr tedy podléhá mému osobnímu vnímání podobnosti, a proto se může váš názor lišit.

Nejprve jsem zcela náhodně zvolil hodnoty 5, 20 a 50, na kterých jsem otestoval svou sadu obrázků. Díky tomu jsem zjistil, že hodnota 50 je příliš vysoká a algoritmus porovnávání je díky ní příliš striktní při vyhodnocování. Naopak hodnoty 5 a 20 vracely mnohem obecnější výsledky, což jsem shledával vhodnějším. Rozhodl jsem se tedy další testy provádět pouze na intervalu $< 5, 20 >$. Při pokusu o změnu počtu pouze o jednu hodnotu jsem pozoroval velmi malou odchylku od původních výsledků, a proto jsem se rozhodl rovnou testovat hodnoty z intervalu s krokem 2. Dále na mne působily výsledky při testování hodnoty 20 přesvědčivěji, a proto jsem začal zkoušet hodnoty od horní hranice sestupně.

K mému překvapení jsem zjistil, že hodnoty 20, 18, 16 a 14 vracely velmi podobné výsledky a rozdíly mezi jednotlivými kroky byly minimální. Sada nevyhovujících obrázků se však postupně zmenšovala. Výraznější rozdíl jsem zaznamenal až u hodnot 12, 10 a 8, které vracely velmi dobré výsledky. Pro kontrolu jsem ověřil i hodnotu 6, která však vracela již velmi obecné výsledky, podobně jako hodnota 5. Ze tří nejlepších kandidátů jsem nakonec zvolil hodnotu 10, která dle mého subjektivního pocitu odpovídala nejlépe realitě.

Tato hodnota by se dle mého názoru měla vztahovat vždy k praktickému použití tohoto algoritmu a podobné testování by mělo následovat vždy při výraznější změně sady testovaných obrázků. Vzhledem k obecnosti použití mé aplikace nemohu přesněji odladit tento parametr, a proto je osobní vnímání v tomto případě zřejmě nejlepším kritériem.

4.2 Optimalizace rychlosti zpracovávání

Rychlost běhu aplikace i výpočtu nad obrázkem má velký vliv na dojem uživatele při používání. Pokud musí uživatel dlouho na výsledky čekat, s velkou pravděpodobností nebude s aplikací spokojen, a možná ji přestane také používat. Proto jsem strávil velké

množství času přemýšlením, jak a kde bych mohl celý proces urychlit. Optimalizace je na místě i z toho důvodu, že samotný JavaScript je velmi pomalý a pokud aplikaci spustí uživatel na méně výkonném zařízení, výsledný výkon aplikace jde dramaticky dolů.

4.2.1 Optimalizace algoritmu

Normalizace vektoru

Aby vracel algoritmus správné výsledky, měl by být výsledný vektor normalizovaný. Záměrně jsem se však o tomto procesu doposud nezmínil, protože jsem přišel na to, že mohu tento krok zcela vypustit. Důvod je zcela zřejmý. Normalizací transformujeme hodnoty do jednotného intervalu $< 0, 1 >$, díky kterému pak můžeme hodnoty porovnávat, protože jsou ve stejném rozsahu.

Protože jsem však hned na začátku celého procesu vytvořil z každé fotky čtverec o stejném rozlišení, převedl jsem tak každý obrázek na jednotné meze. Obrázek má tedy rozlišení 300 pixelů na šířku i na výšku z čehož vyplývá, že interval má v mém případě rozsah $< 0, 90000 >$, což má stejný význam a dopad na výsledky jako interval $< 0, 1 >$. Vzhledem k tomu, že vyhrazený prostor v paměti pro celočíselný datový typ je většinou stejný nebo menší než datový typ pro desetinná čísla, lze hovořit i o úspoře paměti. Vypuštěním tohoto kroku z celého algoritmu se mi tedy povedlo ušetřit nemalé množství výkonu, které se projeví zejména mobilním uživatelům.

Euklidovská vzdálenost

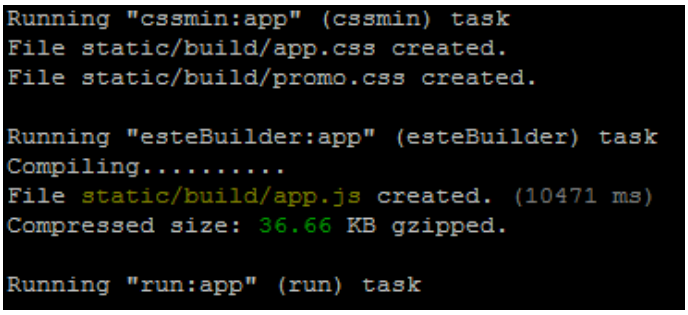
Další prostor pro optimalizaci se mi naskytl při výpočtu euklidovské vzdálenosti mezi jednotlivými dvojicemi porovnávaných vektorů. Tento výpočet obsahuje v posledním kroku druhou odmocninu, která má zajistit uvedení hodnoty zpět z prostoru kvadrátu. Porovnání odmocněných hodnot nebo kvadrátů však výsledek nikterak neovlivní, a proto si mohu dovolit odmocninu z výpočtu zcela odstranit. Vzhledem k tomu, že je třeba provést výpočet pro všechny obrázky v databázi, bude to znamenat velkou úsporu výpočetního výkonu. Protože se tento výpočet nachází na serveru, bude mít tato optimalizace přímý dopad plošně na všechny uživatele ve stejné míře.

4.2.2 Technická optimalizace

Tento typ optimalizace pocítí nejvíce uživatelé méně výkonných zařízení, za které můžeme považovat především mobilní zařízení. U těchto zařízení předpokládáme napájení z baterie, a proto každá úspora výkonu bude mít přímý vliv i na úsporu energie baterie takového zařízení. Closure Compiler v tomto hraje velkou roli, a proto jsem se snažil jeho přínosu využít co nejvíce.

Closure Compiler provádí nejen optimalizace objemu zdrojového kódu, ale také zlepšuje i výkon použitím některých optimalizací. Tyto optimalizace může nejlépe provádět pouze tehdy, pokud mu ke zdrojovému kódu dodám dostatek metainformací. Mezi tyto informace patří například volitelná datová typovost nebo modifikátory přístupu (public, protected, private). Takovýchto informací mu lze předat velké množství. Více o meta-

informacích se lze dočíst v knize Closure: The Definitive Guide [6]. Dalším kritériem pro správnou funkčnost kompilátoru je používání správných zápisů určitých jazykových konstrukcí JavaScriptu. Tento požadavek částečně řeší samotný CoffeeScript, který se snaží generovat JavaScript dle těchto požadavků, avšak programátor je stále nucen psát zdrojový kód striktně dle specifikace. To je však ve výsledku velmi dobře, protože je kód pak dobře čitelný. Osobně to považuji za benefit.



Obrázek 8: Výsledek optimalizace pomocí Google Closure Compileru

Doplnil jsem tedy tyto metainformace všude tam, kde to dávalo smysl a díky tomu se kompilátoru daří zkomprimovat mou aplikaci až na neuvěřitelných 37 KB. V porovnání například s populární knihovnou jQuery v aktuální verzi, která samotná má přibližně 80 KB, se dostávám na polovinu tohoto objemu s celou aplikací. O něco málo lepší je situace s frameworkem AngularJS, který se pyšní stejným objemem, tedy 37 KB. Lze tedy z pohledu objemu aplikace říci, že tam, kde začíná AngularJS, tam končí Google Closure s celou aplikací.

5 Závěr

Řešení tohoto problému mne velmi bavilo a naučil jsem se při tom mnoho zajímavých informací nejen z oblasti analýzy obrazu, ale také z oblasti vývoje webových aplikací. Osobně si myslím, že se mi podařilo stanovený problém vyřešit a velmi rád bych se dále věnoval rozšiřování své aplikace. Nabyté zkušenosti z obou oblastí mohu nyní prakticky aplikovat i na řešení spousty jiných úloh, u kterých mne doposud nenapadlo je řešit pomocí analýzy obrazu. Jako podněty pokračování v mé práci si dovolím uvést několik nápadů, jak by se dala má aplikace ještě zdokonalit:

- Propojení vyhledávání s dalšími reálnými daty podobně, jako v případě Google.
- Přesun výpočtu na grafickou kartu pomocí WebGL nebo WebCL.
- Paralelizace výpočtů pomocí WebWorkers.
- Využití jiné metriky pro porovnávání histogramů nebo kumulativního histogramu.

Michal Šimon

6 Reference

[1] Gonzalez, Rafael C. and Woods, Richard E., *Digital Image Processing (3rd Edition)*, 2007, ISBN 01-316-8728-X.

[2] J.R. Parker, *Algorithms for Image Processing and Computer Vision (second edition)*, 2011, ISBN 978-0470643853.

[3] E. Sojka, *Digitální zpracování a analýza obrazu*, 2000.

[4] Canny, J., *A Computational Approach To Edge Detection, IEEE Trans. Pattern Analysis and Machine Intelligence*, 1986, ISSN 0162-8828.

[5] A. Osmani, *Learning JavaScript Design Patterns*, 2012, ISBN 14-493-3181-5.

[6] M. Bolin, *Closure: The Definitive Guide*, 2010, ISBN 978-1-4493-8187-5.

[7] A. Kaehler, G. Bradski, *Learning OpenCV*, 2013, ISBN 978-144-9314-651.

[8] D. Sanderson, *Programming Google App Engine*, 2012, ISBN 05-965-2272-X.